Erstprüfer/in: Prof. Dr. Ansgar Gerlicher  
Zweitprüfer/in: Sebastian Gerber

**Bachelorarbeit**

im Studiengang  
Medieninformatik

vorgelegt von

**Malte Leon Lohrer**Matr.-Nr.: 27399

am 05. April 2018   
an der Hochschule der Medien Stuttgart

Continuous Integration der Diagnosetoolkette der Daimler AG

# Ehrenwörtliche Erklärung

„Hiermit versichere ich, Malte Leon Lohrer ehrenwörtlich, dass ich die vorliegende Bachelorarbeit (bzw. Masterarbeit) mit dem Titel: „Continuous Integration der Diagnosetoolkette der Daimler AG“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.“

01.04.2018

# Kurzfassung

# Abstract

# Inhaltsverzeichnis

Ehrenwörtliche Erklärung 2

Kurzfassung 3

Abstract 3

Inhaltsverzeichnis 4

Abbildungsverzeichnis 5

Tabellenverzeichnis 5

Abkürzungsverzeichnis 6

Vorwort 7

1 Überblick 8

2 Ziele 9

3 Stand der Technik 11

4 Continuous Integration 12

4.1 Das Konzept der Continuous Integration 12

4.1.1 Gemeinsame Codebasis 13

4.1.2 Automatisierter Build 13

4.1.3 Selbsttestender Build 13

4.1.4 Häufige Integration 14

4.1.5 Builds und Tests nach jeder Änderung 14

4.1.6 Schnelle Build-Zyklen 16

4.1.7 Tests in gespiegelter Produktionsumgebung 16

4.1.8 Einfacher Zugriff auf Build-Ergebnisse 18

4.1.9 Automatisierte Berichte 18

4.1.10 Automatisierte Verteilung 18

Quellenverzeichnis 19

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

CI Continuous Integration

GUI Graphical User Interface (Deutsch: Grafische Benutzeroberfläche)

VCS Version control system (Deutsch: Versionskontrollsystem)

HdM Hochschule der Medien

# Vorwort

# Überblick

# Ziele

Ziel der vorliegenden Arbeit ist es, das Konzept der *Continuous Integration* genau zu beleuchten und theoretischauf die Diagnosetoolkette der Daimler AG anzuwenden. Die dabei gewonnenen Erkenntnisse werden die Vor- und Nachteile einer CI aufzeigen.

Der praktische Teil der Thesis, welcher auf der beiliegenden CD enthalten ist, stellt einen ersten Schritt in Richtung CI der Diagnosetoolkette dar. Ziel dieses Teils ist es, das Diagnosetool *Monaco* automatisiert zu testen um damit die Funktionalität des Tools zuverlässig gewährleisten zu können.

Nachfolgend sind die Feinziele der Praktischen Arbeit aufgelistet:

1. Gemeinsame Codebasis (erfüllt)
   1. Gerade im Hinblick auf eine CI ist es enorm wichtig direkt von Anfang an die Möglichkeit einer gemeinsamen Codebasis zu schaffen. Diese wird durch ein *GitLab Repository* realisiert, welches in Zukunft problemlos von anderen Teams genutzt werden kann.
2. Modularer Code (erfüllt)
   1. Ein weiteres Ziel ist es den Quellcode in möglichst kleine Module aufzuteilen. Das bringt nicht nur durch eine bessere Übersicht Vorteile, sondern kommt auch der Wartbarkeit des Quellcodes enorm entgegen. Außerdem lassen sich kleinere Teile Quellcode deutlich leichter, effizienter und vor allem abgekapselt testen, was bei einer guten CI eine große Rolle spielt.
3. Automatisierung der grafischen Benutzeroberfläche von *Monaco (erfüllt)*
   1. Ein wesentliches Ziel besteht darin, das Tool *Monaco*, auf welche in Kapitel <X> genauer eingegangen wird, zu automatisieren. Da diese Anwendung über eine Grafische Benutzeroberfläche (im Folgenden als „GUI“ bezeichnet) gesteuert wird, müssen hierbei die Aktionen die normalerweise ein Mensch durchführt, vom Code übernommen werden.
4. Bereitstellen von Unit Tests (erfüllt)
   1. Unverzichtbar für eine richtig durchgeführte CI ist es, genügend Tests zu implementieren, welche stets die Funktion der einzelnen Codebestandteile sicherstellen. Hierbei werden sogenannte *Unit Tests* durchgeführt, welche jeder für sich nur sehr kleine des Quellcodes testen, dafür jedoch sehr schnell und präzise.
5. Bereitstellen von Komponenten- und Systemtests (nicht erfüllt)
   1. Anders als bei Unit Tests werden bei Komponententests zusammenhängende Codebereiche getestet. Das sind dann zum Beispiel mehrere Klassen, welche im gesamten Programmablauf zusammenarbeiten.
   2. Eine noch umfassendere Art von Tests stellen Systemtests dar, welche Grundsätzlich das gesamte Produkt testen. Hierbei werden tatsächlich alle Teile der Software im Zusammenspiel getestet. Darunter fallen beispielsweise auch GUI-Tests.
6. Automatisches generieren der Ergebnisse (erfüllt)
   1. Die Nachvollziehbarkeit des Programmdurchlaufs stellt ein weiteres Ziel dar. Diese wird durch das protokollieren der Ergebnisse der einzelnen Teilabschnitte eines jeden Durchlaufs gewährleistet. Nach einem vollständigen Programmablauf werden Daten bereitgestellt, welche genau aufzeigen welche Funktionen zu welchem Zeitpunkt abgelaufen sind und ob es dabei zu Fehlern kam.
7. Abhängigkeiten zwischen den Funktionen klar definieren (erfüllt)
   1. Es muss gewährleistet sein, dass bestimmte Teile des Codes erst dann ausgeführt werden, wenn andere Teile schon erfolgreich durchlaufen wurden. Diese Abhängigkeiten müssen klar definiert werden. Wenn eine Funktion fehlschlägt, werden nachfolgende erst gar nicht ausgeführt. Diese Abhängigkeiten wurden mithilfe von *luigi* realisiert, ein Python Modul auf welches in Kapitel <X> genauer eingegangen wird.
8. Visualisierung des gesamten Programmablaufs (erfüllt)
   1. Eines der bedeutendsten Ziele stellt die Visualisierung des Programmablaufs dar. Wie schon bei der Definition der Abhängigkeiten, wird hier das Python Modul *luigi* genutzt, um stets den Status des Ablaufs in grafisch aufbereiteter Form auf einer Website nachvollziehen zu können.

# Stand der Technik

Hier werde ich die ganze Toolkette von Daimler beschreiben. Wie Läuft Sie ab welche Tools werden benutzt wie genau wird hier gearbeitet – vor allem natürlich in Hinblick auf CI .

Die Toolkette werde ich vermutlich in 3.1, 3.2 , 3.3 usw. unterteilen

# Continuous Integration

Dieses Kapitel befasst sich mit der Continuous Integration als Solche und gibt dem Leser einen Einblick in die Softwareentwicklung unter Anwendung von CI.

Hierfür werden die Grundprinzipien einer CI erläutert und es wird auf deren Vor- und Nachteile eingegangen.

## Das Konzept der Continuous Integration und deren Vorteile

Die Continuous Integration ist nach Wiest (2010, S.13) eine Softwareentwicklungspraktik, welche erstmals als eine der Praktiken der sogenannten Extremprogrammierung (*eXtreme Programming*) populär wurde.

Der Begriff *Continuous Integration* entspringt dem gleichnamigen Artikel von Martin Fowler, welcher das erste Mal im Jahr 2000 erschien und im Jahr 2006 noch einmal überarbeitet und aktualisiert wurde.

In eben diesem Artikel beschreibt Fowler (2006) das Grundkonzept der CI folgendermaßen:

“Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage.”

Grundsätzlich geht es bei CI nach Fowler darum, dass die Softwareentwickler die Änderungen welche sie am Code vornehmen so oft wie möglich integrieren, um die Deltas zwischen der bisherigen und der neuen Version möglichst gering zu halten. Der neue Softwarestand wird dann umgehend gebaut, wodurch Fehler, durch die kleinere Menge an Änderungen im Vergleich zum letzten Softwarestand, viel schneller gefunden werden können. Dadurch soll der Integrationsvorgang selbst zu einem *Non-Event* (dt.: nicht-Ereignis) werden (vgl. Martin Fowler, 2006).

Fowler formulierte in seinem Artikel „Continuous Integration“ außerdem insgesamt 10 Praktiken, welche für eine effektive CI maßgeblich sind.

Für die Namen dieser Praktiken werden im Folgenden die deutschen Übersetzungen von Dr. Simon Wiest (2010: 15f) verwendet.

### Gemeinsame Codebasis

Sämtliche Daten, welche zu einem Softwareprojekt gehören, müssen in einem für alle an dem Projekt beteiligten Akteure offen und jederzeit zugänglich sein. Meist ist hier die Rede von einem Versionskontrollsystem (VCS, engl.: *version control system*). Versionskontrollsysteme gibt es heutzutage zu genüge. Einige der bekanntesten Vertreter sind u.a. *Git*, *Subversion(SVN)* oder auch *Mercurial*.

Es sei an dieser Stelle angemerkt, dass man zwischen drei verschiedenen Arten der Versionskontrollverwaltung unterscheidet: Lokale Versionskontrollsysteme, Zentralisierte Versionskontrollsysteme und Verteilte Versionskontrollsysteme (vgl. Git 19. Oktober 2017, Seite „Getting started - About version control“). Hier und im Folgenden, sind stets verteilte Versionskontrollsysteme (DVCS, engl.: *distributed version control system*) gemeint, wenn von einem Versionskontrollsystem (VCS) die Rede ist.

Die Hauptaufgabe eines VCS besteht darin, jede Änderung an einer Datei genau zu dokumentieren und nachvollziehbar zu machen. Diese Änderungen sind im Nachhinein auf einen genauen Zeitpunkt und auch auf die Person, welche die Änderung vorgenommen hat, zurückzuführen. Dadurch wird paralleles Arbeiten innerhalb desselben Projekts, bzw. sogar innerhalb identischer Dateien, ermöglicht. Entwickler arbeiten, wenn ein VCS benutzt wird, normalerweise auf ihrem eigenen *Branch*[[1]](#footnote-1)und führen einen *commit*[[2]](#footnote-2)aus, sobald sie eine Teilaufgabe abgeschlossen haben und bereit sind, diesen neuen Softwarestand in das VCS zu übertragen. Auf diese Weise können mehrere Entwickler parallel auf ihren jeweiligen *Branches* arbeiten. Zu einem beliebigen späteren Zeitpunkt lassen sich diese Branches zusammenführen. Dies ist eine weitere Aufgabe, die ein VCS übernimmt. Man spricht dann von einem *Merge*, also zu Deutsch einer „Verschmelzung“ von zwei oder mehreren Branches. Es kann dabei zu Konflikten kommen, wenn während des parallelen Entwickelns identische Zeilen des Sourcecodes von mehreren Entwicklern modifiziert wurden. Das VCS benachrichtigt in diesem Moment den Benutzer, welcher den *Merge* durchführen will und bieten die Chance diese Konflikte zu lösen. Es können beliebig viele Branches erstellt werden, welche schlussendlich aber immer mit dem sogenannten Master-Branch zusammengeführt werden. Der Master-Branch sollte stets eine funktionierende Version des Softwareproduktes enthalten, weshalb niemals auf dem Master-Branch selbst entwickelt und getestet wird.

### Automatisierter Build

Das Projekt muss stets aus der Summe seiner Teile vollautomatisch und von Grund auf neu gebaut werden können. Damit ist gemeint, dass zu keiner Zeit Fehlerhafter Quellcode vorliegen darf, welcher verhindern würde, einen Build zu erstellen. Fowler (2006) beschreibt hier zusätzlich, dass es Sinn macht Build-Werkzeuge zu nutzen, welche auf allen nötigen Plattformen zur Verfügung stehen, da Build-Werkzeuge von Entwicklungsumgebungen oftmals proprietäre Dateien beinhalten und damit nicht zwingend auf jedem System funktionieren werden.

### Selbsttestender Build

Während des Build-Prozesses muss das Produkt automatisch getestet werden. Es obliegt dem Entwickler diese Tests sinnvoll zu implementieren. Es ist wünschenswert, so viel Quellcode wie möglich durch Tests abzudecken. Dabei unterscheidet man zwischen

sehr feingranularen Tests wie z.B. den *Unit-Tests*, welche nur sehr kleine, meist unabhängige Abschnitte Testen.

den mittelgroßen *Komponententests* die, wie der Name verrät, ganze Komponenten testen – also beispielsweise die Funktion mehrerer miteinander kommunizierender Klassen.

Den übergreifenden *Systemtests*, welche ein Produkt in seiner Ganzheit testen.

### Häufige Integration

Einer der offensichtlichsten und wichtigsten Punkte dieser Auflistung ist die häufige Integration von Code. Die Entwickler sind dazu angehalten, so oft wie möglich, mindestens jedoch 1 Mal am Tag, Ihren Code in ein Versionskontrollsystem einzuchecken. Simon Wiest (2006: 34) spricht hier von einer kulturellen Veränderung, weil das häufige Einchecken des Quellcodes etwas ist, dass sich viele Entwickler erst angewöhnen müssen, bis es selbstverständlich wird. Bevor eingecheckt werden darf, muss natürlich die Version vorab lokal erfolgreich gebaut werden können um auf diese Weise zu vermeiden, dass Fehlerhafter Quellcode eingecheckt wird. Die Vorteile der häufigen Integration liegen auf der Hand:

Der aktuelle Entwicklungsstand und das aktuelle Master Build liegen nie weit auseinander, was enorm bei der Fehlerfindung hilft, da neue Fehler nur in dem neuen Delta zwischen den beiden Versionen auftreten verursacht werden können.

Die Entwickler können Risikofreudiger entwickeln, da sie jederzeit einen sogenannten *Roll-back* auf den letzten aktuellen Stand durchführen können.

Es kann viel schneller auf Fehler reagiert werden.

Man hat zu jedem Zeitpunkt eines Projekts einen funktionierenden Stand, auch wenn das nicht heißt, dass dieser fertig ist.

### Builds und Tests nach jeder Änderung

Ein absolut essenzieller Punkt einer CI ist, ebenso wie das häufige integrieren von neuem Code, das ebenso häufige bauen und testen des selbigen. Neuer Quellcode sollte bei einer guten CI in der Theorie nach jeder Änderung getestet und anschließend, bei erfolgreichen Tests, gebaut werden. Zu häufiges testen und bauen kann nach Wiest (2010: 38) aber zu einem Rückstau an Builds führen, bzw. zu Zeitintensiv ausfallen. Dr. Simon Wiest schlägt hier als groben Lösungsansatz vor, so oft wie es die Gegebenheiten zulassen zu bauen und nicht Zwingend nach jeder Änderung. Wie oft man den Quellcode tatsächlich kompiliert und baut, hängt letztendlich von individuellen Faktoren ab und muss an diese angepasst werden.

Martin Fowler (2006) beschreibt, dass es Sinnvoll ist einen CI Server zu verwenden, welcher das Versionskontrollsystem überwacht. Wenn ein Projektmitglied einen neuen Stand zu dem verwalteten Repository[[3]](#footnote-3) hinzufügt, bzw. einen sogenannten *commi*tausführt, dann meldet das Verwaltungssystem dem CI Server das und dieses wiederum stößt den Build-Vorgang z.B. auf einem separaten Build-Server an und benachrichtigt das Mitglied über das Ergebnis. Durch die Verwendung eines CI Servers wird sichergestellt, dass nach *jedem* Commit ein Build erzeugt wird und die Entwickler zeitnah die Ergebnisse erhalten.

Abbildung 1: Ablauf der Kontinuierlichen Integration (CI), Quelle: Dr. Simon Wiest "Continous Integration mit Hudson" (2010: 14)

Abb. 1 veranschaulicht diesen Vorgang noch einmal bildlich. Auch wenn nicht zwingend ein CI-Server benötigt wird bietet es sich aus genannten Gründen an einen zu benutzen. Ein CI Server setzt allerdings wiederum schnelle Build-Zyklen voraus. Auch wenn ein CI Server von Grund auf neu entwickelt werden kann, gibt es bereits für alle erdenklichen Ansprüche ausgereifte Tools, welche in den allermeisten Fällen mehr als ausreichend sind (Paul M. Duvall et al, 2011: 85).

### Schnelle Build-Zyklen

Um die Vorteile einer CI voll auszuschöpfen ist, wie in 4.1.5 beschrieben, häufiges bauen essenziell. Es zeigt sich, dass es wichtig ist, die Build-Zyklen so kurz wie möglich zu halten um dadurch die Häufigkeit dieser erst möglich zu machen. Kent Beck (XXX) gibt einen Zeitaufwand von 10 Minuten für einen build als guten Richtwert an. In der Realität ist eine solche Zeit aber nicht immer ohne weiteres erreichbar und bedarf Anpassungen. Abgesehen von Aufrüstungen in Form von leistungsfähigeren Build-Servern, beschreibt Simon Wiest (2010: 39) drei Methoden um Build-Zeiten möglichst kurz zu halten.

1. Staffeln des Builds
   * Der Build selbst wird in mehrere Stufen bzw. kleinere Builds aufgeteilt. Fowler (2006) spricht von einem *Commit-Build* welcher als erstes gebaut wird. Das ist ein kurzer, nur schnelle Unit Tests beinhaltender Build, der zwar nicht komplett ist, dafür aber sehr schnell fertiggestellt werden kann. Dieser Build geht den nachgelagerten Builds voran und ist ausschlaggeben dafür, ob diese nachgelagerten Builds überhaupt erstellt werden oder nicht. Schlägt der *Commit-Build fehl*, werden nachgelagerte Builds meist ga nicht erst gebaut, das spart Zeit und Ressourcen. Außerdem bekommen Entwickler auf diese Weise schneller eine Rückmeldung (Wiest, 2006: 39)
2. Modularisierung
   * Das Zerlegen des Projekts in eigene, unabhängige Module bietet den Vorteil, dass nur diejenigen Module neu gebaut werden müssen, welche auch verändert wurden. Die übrigen Module kann man demnach ganz einfach aus den vorherigen Builds entnehmen und weiterverwenden (Wiest, 2006: 39).
3. Parallelisierung
   * Die Parallelisierung setzt eine Modularisierung voraus. Wenn Module nicht voneinander abhängig sind, spricht nichts dagegen diese parallel auf mehreren Rechnern zur selben Zeit zu bauen (vgl. Wiest 2006: 39).
   * Paul M. Duvall (et al, 2011: 96) empfiehlt das parallelisieren, oder auch *distributed integration* genannt als den letztmöglichen Versuch die Build-Dauer zu verkürzen, weil es ein äußerst komplexer Prozess ist.

### Tests in gespiegelter Produktionsumgebung

Auch wenn es der Idealfall wäre, ein Softwareprodukt stets in der Umgebung zu testen, in der es später tatsächlich eingesetzt wird, ist dies meist aus diversen Gründen nicht möglich. Zum einen spielen hier kosten eine Rolle, weil es schlicht unwirtschaftlich sein kann, extrem große Datenmengen zu transferieren oder aber extra dafür neue oder spezielle Hardware kaufen zu müssen. Zum anderen kommt ein Softwareprodukt in seinem Einsatzbereich nicht selten in Kontakt mit personenbezogenen Daten, welche aus juristischen Gründen nicht an Dritte weitergereicht werden dürfen. Das schließt auch Entwickler dieser Software mit ein, welche die Software nur im Auftrag entwickeln. Ein weiterer Grund kann sein, dass die Produktionsumgebung noch nicht existiert, weil sie sich selbst noch in Entwicklung befindet (Wiest, 2006: 40).

Auch wenn es durch diese und weitere Gründe nicht immer möglich sein wird, die Produktionsumgebung exakt abzubilden, sollte trotzdem versucht werden, eine Umgebung zu schaffen, welche der Originalen so nah wie möglich kommt. Dadurch können Fehler frühzeitig erkannt werden, welche unter Umständen nur in dieser einen speziellen Umgebung auftreten. Solche Fehler lassen sich demnach in einer schlechter abgebildeten Produktionsumgebung erst gar nicht auffinden bzw. reproduzieren und können im Nachhinein zu deutlich mehr Aufwand und Kosten führen, wie es die Abbildung einer genaueren Umgebung getan hätte.

Paul M. Duvall (et al, 2011: 194) rät dazu die Entwicklungsumgebung von Anfang an absolut sauber und in fein unterteilen Ebenen aufzubauen. Er empfiehlt folgende Ebenen nacheinander aufzubauen:

* Das Betriebssystem installieren
* Betriebssystem konfigurieren (z.B. Netzwerkeinstellungen, Benutzerkonten, Firewall-Einstellungen)
* Server Komponenten bereitstellen
* Server konfigurieren
* Software von Drittanbietern aufspielen
* Maßgeschneiderte Software installieren/Entwickeln (Das eigentliche Produkt)

Auf diese Weise kann man Fehlerquellen ausschließen, welche z.B. durch Fremdsoftware oder unbekannte Konfigurationen ausgelöst würden.

Simon Wiest (2006: 40) empfiehlt an dieser Stelle als etwas komfortableren Lösungsansatz *Cloud-Computing*[[4]](#footnote-4). Hierbei werden gewünschte Hardwarekonfigurationen über das Internet bereitgestellt und müssen somit nicht neu gekauft werden. Wenn man ganze Systeme bzw. Umgebungen über diesen Service bereitgestellt bekommen spricht man auch von *Platform as a Service* (PaaS). Für die Dauer der Nutzung des Services fallen selbstverständlich auch Kosten an. Da man diesen Service allerdings nur für einen absehbaren Zeitraum in Anspruch nimmt, liegen diese Kosten sehr wahrscheinlich unter den Kosten, welche andernfalls für die Neuanschaffung einer Nachbildung der gesamten Produktionsumgebung anfallen würden.

Letztendlich muss hier individuell entschieden werden, da es sich nicht pauschal beantworten lässt, welche der beiden Varianten die wirtschaftlichere ist.

### Einfacher Zugriff auf Build-Ergebnisse

Gerade die Verwendung einer Versionsverwaltung erleichtert den Zugriff auf sämtliche Softwarestände ungemein. Beteiligte an einem Projekt, seien es Entwickler, Teamleiter, Tester oder Kunden, bekommen mit Einführung eines CI-Systems (welches eine Versionsverwaltung einschließt) stets sämtliche Builds an einem zentralen Ort bereitgestellt. Dadurch wird laut Simon Wiest (2010: 40) die Wahrscheinlichkeit erhöht, das neue Softwarestände schneller zu Beteiligten gelangt, welche wiederum schneller Rückmeldung über die Qualität der jeweiligen Stände geben können.

### Automatisierte Berichte

Da ein CI-System stets den Überblick bzw. die Kontrolle über sämtliche Build-Prozesse hat lassen sich auf diese Weise sehr genau alle Schritte und Ergebnisse nachvollziehen. Diese Ergebnisse können gezielt an betroffene Entwickler gesendet werden. Dieser Ansatz wird auch *Aktive Benachrichtigung* genannt, da in diesem Fall das CI-System aktiv Nachrichten versendet und die Betroffenen Personen nicht selbst nach diesen Informationen suchen müssen. Diese Aktiven Benachrichtigungen sollten sehr detaillierte Informationen enthalten, welche z.B. die Auslöser eines Fehlers eingrenzen. Außerdem sollten Entwickler und andere Beteiligte nur bei wichtigen Ergebnissen benachrichtigt werden, um ein Stören dieser Funktion zu vermeiden (vgl. Wiest 2010: 41).

### Automatisierte Verteilung

Im letzten Schritt kann sich ein CI-System selbst noch um das verteilen eines Produktes kümmern. Man spricht hierbei von *continuous deployment*. Heutzutage gibt es Unternehmen, welche neue Softwarestände teils *mehrmals täglich* ausbringen. Paul M. Duvall (2011: 190) bezeichnet eBay, Amazon & Google als Vorzeigebeispiele in dieser Hinsicht.

Durch die bisherigen Schritte einer gut durchgeführten CI ist es möglich jederzeit funktionierende Software zu verteilen weil diese, wie es der Name verrät, kontinuierlich Integriert und bei jedem neuen Build getestet, und nur dann freigegeben wird, wenn sie sich Fehlerfrei bauen lässt.

Als besonders komfortabel ist an dieser Stelle die *Roll-back[[5]](#footnote-5)* Funktion hervorzuheben. Da ältere Softwarestände, wie schon mehrfach erwähnt, selbstverständlich archiviert werden, ist man durch ein CI-System stets in der Lage die Ausbringung einer Neuen Software Version rückgängig zu machen, indem man einfach einen *Roll-back* auf einen älteren Stand vollzieht. Von diesem älteren Stand ist bereits bekannt, dass er fehlerfrei läuft, da er logischerweise bereits im Einsatz war. Das führt wiederum dazu, dass Entwicklern mutiger agieren können, weil das CI System ihnen jederzeit ein virtuelles Fallnetz bietet.

## Nachteile der CI

Nachdem viele Vorteile einer gut durchgeführten CI im vorangegangenen Kapitel beleuchtet wurden soll an dieser Stelle auch auf die Nachteile, welche die Einführung einer CI mit sich bringen kann, eingegangen werden.

* Einige der beschriebenen Vorteile, wenn nicht sogar die meisten, werden nur dann zu bemerken sein, wenn die CI kompromisslos durchgeführt wird. Das hat zur Folge, dass *alle* Beteiligten Personen stets alle Praktiken bzw. Prinzipien, welche die CI fordert, einhalten. Diese Art zu Entwickeln wird für viele dieser Personen vermutlich eine Umstellung ihrer Arbeitsweise erfordern und wird sich nicht über Nacht einstellen. Diese Umstellung kann also relativ viel Zeit in Anspruch nehmen.
* Eine gemeinsame Codebasis bildet einen unverzichtbaren Teil der CI und setzt voraus, dass die Entwickler absolut gewissenhaft mit ihrem Code umgehen. Es sollte wie schon beschrieben nur Code in das VCS eingecheckt werden, welcher getestet wurde und stabil läuft. Es besteht die Gefahr, dass fehlerbehafteter Code durch unzureichende Testabdeckung nicht als solcher erkannt und in das VCS gelangt. Dadurch kann es im späteren Verlauf eines Projektes zu Fehlern kommen, welche sich nicht mehr so einfach finden lassen, da diese sich *nicht mehr* nur in dem kleinen Delta zwischen der vorangegangenen und der aktuellsten Version befinden können, sondern gegebenenfalls in Teilen von deutlich älteren Softwareständen verstecken.
* Ein CI-System wird nur so gut sein wie die Tests es zulassen. Da das CI-System nicht selbst den Sourcecode testet, sondern lediglich die Tests anstößt, welche die Entwickler zur Verfügung gestellt haben, liegt es letztendlich an diesen Tests und damit an den Entwicklern, wie gut eine CI funktioniert oder auch nicht. Diese Tests sollten, wie bereits in 4.1.3 erwähnt, so viel Sourcecode wie möglich abdecken. Aber wie beschrieben, sollte nicht nur der Sourcecode an sich, bzw. dessen Funktionen getestet werden, sondern ebenso sollte auf höheren Ebenen durch Komponenten- bzw. Systemtests ein Zusammenspiel dieser Fehler geprüft werden können. Diese Tests zu erstellen Bedarf wiederum viel Zeit.
* Da ein CI-System in den meisten Fällen zusätzliche Hardware in Form eines CI-Servers, eines Build-Servers und eventuell auch einem eigenen Server auf den ein Versionskontrollsystem läuft besteht, entstehen hier Initialkosten welche je nach Projektumfang variieren. Große Projekte, welche deutlich mehr Zeit benötigen um gebaut zu werden, setzen wahrscheinlich potentere Hardware bzw. sogar mehr Hardware (im Falle einer Parallelisierung des Build Prozesses) voraus als es kleinere würden.
* Da häufig Integriert wird, wird ebenso häufig gebaut, was Zeit in Anspruch nimmt. Auch wenn die Zeit, die ein Build benötigt um erstellt zu werden, wie in 4.1.6 erläutert, verkürzt werden kann, so kann es sehr Aufwendig sein diese Optimierungen vorzunehmen und letztendlich sind diese keine Garantie für die gewünschten Ergebnisse. Wenn die Build Zeiten aus welchen Gründen auch immer nicht kurz genug gehalten werden können, kann es wegen eines Rückstaus an Builds (vgl. Wiest 2010: 38) zu einer Verzögerung des gesamten Projektzeitplans kommen, weil Entwickler auf die Ergebnisse der Builds warten müssen.

Die überwiegende Mehrheit der Nachteile, welche die Continuous Integration verursachen kann, werden nur durch den Umgang mit selbiger ausgelöst – Also durch die Entwickler. Es gibt wie beschrieben auch solche, die nicht von den Entwicklern, bzw. den Beteiligten abhängen, wie zum Beispiel Initialkosten durch Neuanschaffung von Hardware, dennoch lässt sich sagen, dass die meisten Nachteile bei gewissenhafter und korrekter Durchführung nicht auftreten sollten. Diese korrekte Durchführung muss natürlich erst einmal zur Routine werden, was Zeit und Erfahrung bedarf. Trotzdem überwiegen die Vorteile der CI klar deren Nachteilen, vor allem dann, wenn die Nachteile durch eine korrekt angewendete CI erst gar nicht auftreten.

# Eingeschlagener Realisierungsweg

Dieses Kapitel beschäftigt sich mit dem praktischen Teil der Bachelor Arbeit. Das übergeordnete Ziel war das Tool *Monaco*, welches innerhalb der Diagnosetoolkette zum Einsatz kommt, automatisiert zu testen und dem Anwender die Testergebnisse, in visuell und strukturell aufbereiteter Form, zur Verfügung zu stellen. Die feiner definierten Ziele sind in Kapitel 2 „Ziele“ zu nachzulesen.

## Verwendete Tools & Softwaretechnologien

In den folgenden Unterkapiteln wird genauer auf die Tools & Softwaretechnologien eingegangen, welche zur Erstellung des Programms benutzt wurden.

### DTS Monaco

### Python

Der gesamte Source-Code wurde in mithilfe der Programmiersprache *Python* geschrieben. Python ist eine interpretierte, objektorientierte Programmiersprache (vgl. Python.org). Eine Interpretierte Programmiersprache hat den Vorteil, dass sie nicht erst durch einen *Compiler*[[6]](#footnote-6)in eine für die jeweilige Plattform ausführbare Datei übersetzt werden muss, sondern direkt von einem *Interpreter*[[7]](#footnote-7)analysiert und ausgeführt werden kann. Es sei gesagt, dass dafür zwar das eigentliche Ausführen des Quellcodes länger dauert als das Ausführen von bereits kompiliertem Code, allerdings ist der Quellcode eines in Python verfassten Programms dafür meistens recht kurz. Dadurch wird Zeit während der Entwicklung gespart. Eine Interpretierte Programmiersprache bringt allerdings auch die Gefahr mit sich, dass zur Laufzeit Fehler auftreten können, welche beim Entwickeln mit einer anderen, *Compiler*-basierten Programmiersprachen, wie z.B. *C/C++* spätestens beim Kompilieren aufgefallen wären. Der Interpreter unterstützt den Entwickler allerdings ausreichend bei der Fehlersuche, indem er eine *Exception*[[8]](#footnote-8) anzeigt und die Programmausführung an dieser Stelle abbricht*.* Das grenzt den Bereich, in dem es zu einem Fehler gekommen ist, stark ein und vereinfacht die Suche nach diesem, sodass der Nachteil weniger stark ins Gewicht fällt als angenommen werden könnte.

Mit Python lassen sich im Vergleich zu anderen Programmiersprachen deutlich schneller Ergebnisse erzielen. Einer der Gründe warum das so ist, liegt in der sehr einfach gehaltenen Syntax des Codes. Statt, wie es zum Beispiel in Java der Fall ist, einzelne Code-Blöcke in geschweiften Klammern zu schreiben, geschieht diese Form der „Trennung“ in Python durch einfaches Einrücken des Codes, welcher sich in einem Block befindet. Statt das Ende einer Zeile z.B. mit einem Semikolon zu versehen, übernimmt diese Funktion in Python ein einfacher Zeilenumbruch. Zugegeben sind das nur Feinheiten, allerdings summieren sich diese bei mehreren tausend Zeilen Code schnell zu einer beträchtlichen Summe an gesparter Zeit zusammen. Des Weiteren kommt Python mit einer sehr reichhaltigen und mächtigen *Library*[[9]](#footnote-9)daher, die das Entwickeln von neuem Quellcode oft stark vereinfacht und vorantreibt. Gerade diese reichhaltige Bibliothek an Funktionen hat bei der Entwicklung des Projekts enorm geholfen.

Es gibt mittlerweile einige unterschiedliche Python Distributionen, welche sich vor allem in der nativ vorhandenen Auswahl an vorinstallierten Modulen bzw. ihrer Library unterscheiden. Während diesem Projekt wurde ausschließlich mit der Python Distribution *Anaconda* entwickelt auf welche im Folgenden eingegangen wird.

### Anaconda

*Anaconda* ist eine relative junge Python Distribution und war früher unter dem Namen „Continuum Analytics“ bekannt. Eine Python Distribution enthält neben dem eigentlichen Interpreter meist auch nützliche P*ackages[[10]](#footnote-10)* und Werkzeuge oder sogar Entwicklungsumgebungen um mit der Programmiersprache zu arbeiten. Wie schon erwähnt beinhaltet *Anaconda* eine große Library welche über 100 Packages beinhaltet, wobei es die Möglichkeit gibt unzählige weiter zu installieren. An dieser Stelle kommt der Package Manager *Conda* ins Spiel mit welchem dies äußerst komfortabel gelöst wird. Mit *Conda* lassen sich bequem weiteren Packages suchen und installieren. Diese Packages kommen aus dem Hauseigenen Repository *repo.continuum.io*, was den Vorteil mit sich bringt, dass Anaconda® diese Packages selbst überwacht bzw. wartet und auf dem aktuellsten Stand hält. Somit ist sichergestellt, dass keine veralteten Packages in ein Projekt gelangen, welche unter Umständen Schwierigkeiten verursachen.

Es sei gesagt, dass es kein *Anaconda* braucht, um Packages zu installieren oder um generell mit Python zu arbeiten. Dennoch bietet eine Distribution wie *Anaconda* dem Entwickler Vorteile - meist in Form von Komfortfunktionen - welche den Arbeitsalltag und damit das Entwickeln mit Python erheblich erleichtern können.

### PyCharm

### GitLab

### Unit Tests

### Luigi

# Quellenverzeichnis

**Dr. Simon Wiest** (2010): Continuous Integration mit Hudson, Grundladen und Praxiswissen für Einsteiger und Umsteiger. Heidelberg: dpunkt.Verlag

**Martin Fowler** (2006): Continuous Integration. https://www.martinfowler.com/articles/continuousIntegration.html (23. Februar 2018)

**Wikipedia, Die freie Enzyklopädie** (21. Juni 2017): Seite „Repository“. <https://de.wikipedia.org/w/index.php?title=Repository&oldid=166597004> (26. Februar 2018)

**Wikipedia, Die freie Enzyklopädie** (18. August 2015): Seite „Commit“. <https://de.wikipedia.org/w/index.php?title=Commit&oldid=145145351> (26. Februar 2018)

**Paul M. Duvall** (August 2011): Continuous Integration – improving software quality and reducing risk. Crawfordsville, Indiana, 6. Auflage: Addison-Wesley, Pearson Education

**Wikipedia, Die freie Enzyklopädie** (1. Februar 2018): Seite „Cloud Computing“. <https://de.wikipedia.org/w/index.php?title=Cloud_Computing&oldid=173573628> (27. Februar 2018)

**Git** (19. Oktober 2017): Seite „1.1 Getting Started - About Version Control“. <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control> (01.03.2018)

1. Als einen *Branch* (dt.: Zweig) bezeichnet man die Abspaltung einer anderen Version innerhalb innerhalb einers [↑](#footnote-ref-1)
2. *Commit* ist ein Ausdruck aus der [Softwaretechnik](https://de.wikipedia.org/wiki/Softwaretechnik), der die bestätigende [Freischaltung](https://de.wikipedia.org/wiki/Freischaltung) einer Änderung beschreibt (Wikipedia 18. August 2015, Seite „Commit“) [↑](#footnote-ref-2)
3. Ein Repository ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung von digitalen Objekten für ein digitales Archiv (Wikipedia 21. Juni 2017, Seite „Repository“) [↑](#footnote-ref-3)
4. „Cloud-Computing beschreibt die Bereitstellung von IT-Infrastruktur […] über das Internet“ (Wikipedia 01. Februar 2018, Seite „Cloud Computing“) [↑](#footnote-ref-4)
5. Der *roll-back* beschreibt in der Informatik den Vorgang des „Zurücksetzens“ von bestimmten Vorgängen. In diesem Fall das Zurücksetzen eines Softwarestandes. [↑](#footnote-ref-5)
6. Ein *Compiler* ist ein Computerprogramm, welches Quellcode in eine für die auszuführende Plattform lesbare Form übersetzt, sodass diese Plattform den Code schneller ausführen kann. (Wiki nachschauen) [↑](#footnote-ref-6)
7. Ein *Interpreter* liest Quellcode ein, analysiert diesen und führt ihn anschließend direkt aus ohne ihn vorher zu Kompilieren. (wiki nachschauen) [↑](#footnote-ref-7)
8. Eine *Exception* (dt.: Ausnahme) signalisiert, dass es bei der Ausführung von Quellcode zu einem Fehler kam. Das Programm wird normalerweise an dieser Stelle abgebrochen, außer eine Exception wird vom Entwickler ausdrücklich erwartet und ignoriert. [↑](#footnote-ref-8)
9. Eine *Library* (dt.: Bibliothek) stellt in der Informatik eine Sammlung an Funktionen und Routinen bereit um diese im eigenen Quellcode verwenden zu können. [↑](#footnote-ref-9)
10. Ein *package* (dt.: Paket) stellt in Python eine Möglichkeit dar, Module zu strukturieren. Ein *Package* umfasst in der Regel ein oder mehrere Module und kann auch weitere *packages* enthalten. Das *Package* selbst ist im Grunde genommen ein normaler Ordner der zwingend eine Python Datei enthalten muss, welche „*\_\_init\_\_.py*“ genannt werden muss. [↑](#footnote-ref-10)